Revisiting the Majority Problem: Average-Case Analysis with Arbitrarily Many Colours

Anthony Kleerekoper[1]

1 **School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University,UK**
a.kleerekoper@mmu.ac.uk

Abstract

The majority problem is a special case of the heavy hitters problem. Whilst the special case of two-colours has been well studied, the average-case performance for arbitrarily many colours has not. In this paper we present an analysis of the average-case performance of the three deterministic algorithms that appear in the literature. For each, we consider straightforward optimisations that, to our knowledge, have not been previously considered.

1998 ACM Subject Classification F.2.2 [Analysis of Algorithms and Problem Complexity] Non-numerical Algorithms and Problems

Keywords and phrases average-case analysis, data streams, heavy hitters, majority problem

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The majority problem is a special case of the heavy hitters problem. Indeed, research into the more general case was prompted by one of the solutions to the majority problem (see [4], Sec. 5.8 and [10]).

Given a collection of balls, $\{x_1, ...x_n\}$, each of which is coloured with one of m colours, the majority problem is to produce a ball with the same colour as more than 50% of the balls, or to state that no such ball exists.

Boyer and Moore were the first to propose a solution which they called MJRTY [4]. Soon after, Fischer and Salzberg proved that when there are arbitrarily many colours $\lceil \frac{3n}{2} \rceil - 2$ comparisons are necessary and sufficient [7]. They provided an algorithm with that worst-case performance. Some time later, Matula repeated the proof independently and proposed an alternative algorithm that has since been called the Tournament algorithm [9].

Aside from the proofs for the worst-case, all the analysis of the majority problem has been for the special case where there are only two possible colours. In that case it has been shown that $n - \nu(n)$ comparisons are necessary and sufficient, where $\nu(n)$ is the number of 1s in the binary expansion of n [11, 2, 12]. Assuming that all 2^n possible inputs are equally probable, the average-case complexity has been shown to be lower-bounded by $\frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + O(\log n)$ [3]. Under the same assumption, it has been shown that the average-case complexity of Boyer and Moore's MJRTY algorithm is $n - \sqrt{2n/\pi} + O(1)$ [1].

In this paper, we consider the average-case complexity of the three deterministic algorithms and analyse the expected number of comparisons assuming that there are an arbitrary number of colours and these colours are uniformly distributed among the balls. For each of the algorithms we note straightforward optimisations and analyse their effects.

We present our analysis in terms of the plurality colour (the colour that appears with the highest frequency) which may or may not be a majority. This is to enable a more

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

straightforward comparison of the algorithms without reference to the number of colours or their distributions.

2 Boyer and Moore's MJRTY Algorithm

2.1 Brief Description of the Algorithm

Boyer and Moore's algorithm, MJRTY, is based on the concept of pairing off balls. They note that if a majority exists it is possible to pair every majority-coloured ball with one other ball of a different colour, discard those pairs and be left with at least one unpaired ball of the majority colour. If there is no majority, however, this process may leave an unpaired ball of an arbitrary colour and so a second, verification phase is needed.

During the first phase two variables are needed - one to store the index of the candidate ball (whose colour is candidate for the majority) and one to store the counter for the candidate which is initially zero. Whenever the candidate counter is zero the next ball drawn becomes the candidate colour; otherwise the ball that is drawn is compared to the candidate ball. If the two balls are the same colour then the candidate counter is incremented otherwise it is decremented. Incrementing the candidate is a way of storing the fact that we have seen more balls of that colour and decrementing the counter is a way of pairing off and discarding those pairs.

The second phase performs a linear scan counting the number of balls that match the candidate colour. If more than half the balls match then there is a majority, otherwise there is not.

The pseudocode for the algorithm is in Appendix A.

2.2 Analysis of the Algorithm

In the first phase we compare balls every time we go through the loop except when the counter is zero. Calculating the expected number of comparisons is therefore the problem of calculating the expected number of times the counter will be zero. In order to present our analysis in terms of the plurality colour we assume that whenever the counter is zero, the next ball drawn will be of the plurality colour. This gives a lower-bound on the number of times the counter will be zero and therefore an upper-bound on the number of comparisons.

To analyse the number of times the counter will be zero we consider the expected number of balls that are drawn between the counter being zero and returning to zero. Suppose that a ball has been drawn, then the counter is now equal to one and we must continue comparing balls until we have drawn one more non-matching ball than the matching balls.

Let p be the proportion of balls of the plurality colour, then after T draws we will have drawn Tp balls of the plurality colour (which we assume matches the candidate) and $T(1-p)$ non-matching balls. The number of draws we have to make before having one more non-matching ball is the solution to the equation $T(1-p) - Tp = 1$. That is, $T = 1/(1-2p)$ and the number of draws between consecutive zeros is $1 + 1/(1-2p)$. Clearly this has no sensible solution when $p \geq 0.5$ and therefore when there is a majority we would expect that once a ball is drawn the counter never returns to zero.

If the number of draws between consecutive zeros is $1 + 1/(1-2p)$ then the expected number of times the counter is zero is $1 + n(1-2p)/2(1-p)$ and the expected number of comparisons is $n - 1 - n(1-2p)/2(1-p) = n/(2-2p) - 1$.

During the second phase the algorithm will perform comparisons until the counter reaches $n/2 + 1$. When there is no majority this will never happen and we will have n

comparisons. When there is a majority then, by the uniformity assumption, this will happen when $Tp = n/2 + 1$ which is when $T = (n + 2)/2p$.

Overall, then, the expected number of comparisons is:

$$E[C] = \begin{cases} \leq \dfrac{n}{2-2p} - 1 + n & \text{if } p < 0.5 \\ \\ n - 1 + \dfrac{n+2}{2p} & \text{otherwise} \end{cases} \tag{1}$$

2.3 Optimisations and their Analysis

Here we consider three straightforward optimisations and analyse their impact on the number of comparisons.

The first improvement is to stop the first phase if the candidate counter reaches $n/2 + 1$ and then the second phase is not needed at all. From our uniformity assumption we can say that we expect Tp balls to be drawn from the majority colour and $T(1-p)$ to be of the other colours. The candidate counter would therefore be expected to be $T(2p-1)$. Setting this to $n/2 + 1$ gives the expected number of balls that must be seen before the candidate counter reaches $n/2 + 1$ which is:

$$E[T] = \frac{n}{2(2p-1)} + \frac{1}{2p-1}$$

The maximum number of balls that can be drawn is obviously n and so the minimum value of p for which we can expect this improvement to apply can be found by resolving $n/2(2p-1) + 1/(2p-1) = n$. This gives a value of $p = 3/4 + 1/2n \approx 3/4$, for large n.

If we modify the algorithm to retain a counter for the number of remaining balls, then two other early stopping conditions apply. Firstly, the first phase can stop if the candidate counter is larger than the number of remaining balls since then the candidate will not change.

Here we again utilise the fact that after T balls the candidate counter would be $T(2p-1)$ and this time set that equal to the number of remaining balls which is $n - T$. This gives the expected number of balls that must be drawn before the condition becomes active, namely $n/2p$.

This improvement potentially conflicts with the more powerful one introduced earlier because it could end the first phase before the counter reaches $n/2 + 1$ and necessitate a second phase when otherwise no second phase would be needed. We have already seen that if $p \gtrapprox 0.75$ we can expect the candidate counter to reach $n/2 + 1$. Therefore, the MJRTY algorithm must be modified to approximate p during the first phase by using the assumption that the candidate counter $c = T(2p-1)$. For the analysis we apply the first stopping condition when $p \gtrapprox 0.75$ and the second condition when $0.5 < p \lessapprox 0.75$.

A second improvement available by counting the remaining number of balls is to end the second phase if the counter plus the number of remaining balls is less than $n/2 + 1$ because then the candidate is certainly not the majority.

Since there is no majority we cannot predict which ball will be chosen as the candidate but we can provide an upper-bound on the expected number of comparisons by assuming that the plurality colour is chosen. In this case after seeing T balls the candidate counter will be Tp and we add $n - T$ and set to $n/2 + 1$ to give the number of balls processed before the

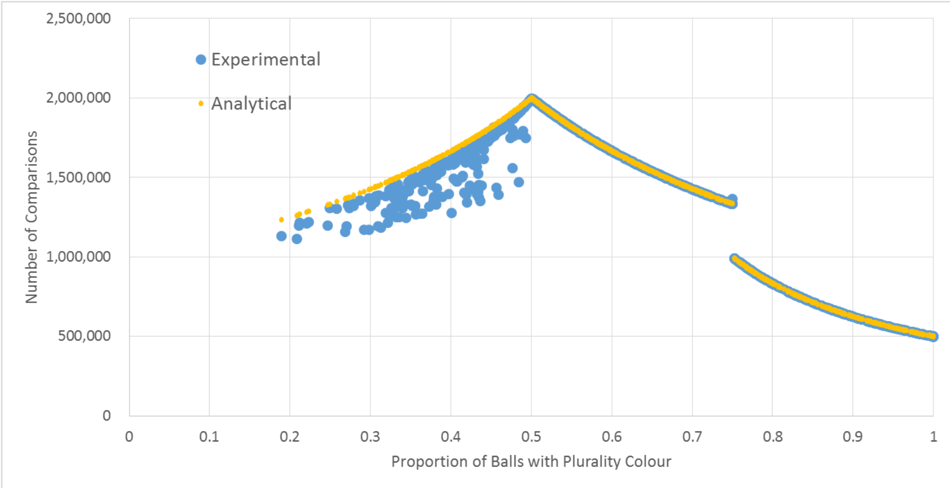

Figure 1 A comparison of the expected and actual number of comparisons for the MJRTY algorithm with all the optimisations validates our analysis, providing an upper bound for $p < 0.5$.

condition activates which is $n/(2-2p)$. Combining all of these conditions gives the overall expected number of comparisons:

$$E[T] = \begin{cases} \leq \dfrac{2n}{2-2p} - 1 & \text{if } p \leq 0.5 \\[2ex] \dfrac{n+1}{2p} + \dfrac{n}{2p} & \text{if } 0.5 < p \lessapprox 0.75 \\[2ex] \dfrac{n}{2(2p-1)} + \dfrac{1}{2p-1} & \text{otherwise} \end{cases} \tag{2}$$

2.4 Experimental Validation

We validate our analysis with experiments in which we create 1,000 arrays each of 1,000,000 items. For each array a set of proportions is generated by iteratively selecting a proportion, q_i, at random from a uniform distribution in the range $(0, r)$ where r is $1 - \sum q_i$, i.e. the remaining proportion available. The process continues until r becomes less than 10% at which point the final proportion is taken as simply $1 - r$. Once the proportions have been generated, an array is created with colours chosen in accordance with the proportions.

This generation method gives an arbitrary number of colours (depending on the proportions selected). The final array is randomly shuffled such that the colours are approximately uniformly distributed throughout the array.

The results in Fig. 1 show the different behaviour of the MJRTY algorithm in the three regions depending on p. Experimental results validated our analysis with the average discrepancy between our analysis and experimental results being just 2.1% with a standard deviation of 5.0%. This is obviously skewed by the values where $p < 0.5$ where our analysis provides only an upper bound. In the range of $p > 0.5$, the average error falls to 0.04% with a standard deviation of 0.1%.

3 Fischer and Salzberg's Algorithm

3.1 Brief Description of the Algorithm

The idea behind the Fischer-Sazlberg algorithm is that, if there is a majority, it is impossible to order all the balls without at least one pair of adjacent balls having the same colour [1]. Their algorithm therefore attempts to produce such an arrangement and fails to do so if a majority colour exists.

There are two phases to the algorithm. During the first phase the balls are drawn and placed either in the list or the bucket. If the ball is the same colour as the last ball in the list then it is placed in the bucket. Otherwise it is added to the end of the list and a ball from the bucket is then added after it if there are any balls in the bucket. The final ball in the list at the end of the first phase is the candidate ball and will be the same colour as all the balls in the bucket if there are any. At this point, the list has an ordering without adjacent balls of the same colour. The second phase serves to check whether the balls from the bucket can be inserted into that ordering without breaking the adjacency constraint.

During the second phase the list is traversed in reverse order with the last ball in the list compared to the candidate. If the two are the same then the last two balls in the list are discarded because balls from the bucket cannot be inserted aither at the end of the list or between the last two balls. If they are not the same then the last ball in the list and one from the bucket are discarded because a ball from the bucket can be inserted at the end of the list without breaking the adjacency constraint. The algorithm stops if a ball is ever needed from the bucket but the bucket is empty because then we have successfully found an ordering that does not break the adjacency constraint and there is no majority colour. If the entire list is traversed and balls remain in the bucket then any of those balls is of the majority colour.

The pseudocode for the algorithm is in Appendix B.

3.2 Analysis of the Algorithm

The original algorithm will always use $n - 1$ comparisons in the first phase but the second phase will depend on whether or not there is a majority. If there is a majority then the entire list is traversed two balls at a time and so the number of comparisons is half the number of balls in the list.

Following our uniformity assumption we would expect that every second ball in the list is of the majority colour. Therefore, the length of the list is determined by the number of non-majority coloured balls which is $n(1 - p)$. The total number in the list is twice this but we traverse the list two balls at a time so the expected number of comparisons in the second phase is $n(1 - p)$.

When there is no majority we would expect the bucket to be empty after the first phase and the second phase would end as soon as an item is encountered in the list that does not match the candidate. If $p = 0.5$ then we would expect every second item in the list to be of the same colour and need to traverse the entire list before finishing. If the number of balls in the plurality were one less than half then we would expect one pair of balls that do not contain the plurality colour and which would trigger the stopping condition when encountered. By the uniformity assumption we would expect this pair to be in the middle of the list. If there were two balls fewer than half then we would expect to have two pairs of

[1] There is one exception, in the case that n is odd and the first, last and every second ball is of the same colour. We do not consider this case here for simplicity

non-plurality colours set at a third and two-thirds of the way through the list. In general, if there are $n/2 - x$ balls of the plurality colour then we would expect to require $n/2(x+1)$ operations in the second phase. Since we know that there are, in fact, pn balls of the plurality colour we can give the expected number of comparisons in the second phase:

$$E[T_2] = \frac{n}{2n(0.5-p)+2}$$

3.3 Optimisations and their Analysis

If we add a counter to the bucket, we can provide two early stopping conditions. The first is triggered if the bucket ever contains more than half the balls in which case we know it contains the majority balls and do not even need the second phase. If, during the second phase, we also record the number of balls remaining in the list then we can end the second phase early if there are more balls in the bucket than in the list as we can be sure that the bucket contains the majority balls.

The first stopping condition can only apply when there is a majority. Since we are assuming that the balls are drawn uniformly we would expect that every second ball in the list is from the majority with some other colour in between. After T balls are drawn the expected length of the list is $2T(1-p)$ which means that the expected size of the bucket is $T(2p-1)$. From this we can derive the expected number of balls that must be drawn before the bucket contains $n/2+1$ balls:

$$E[T] = \frac{n}{2(2p-1)} + \frac{1}{2p-1}$$

The maximum number balls that can be drawn is obviously n and so the minimum value of p for which we can expect the first condition to apply can be found by resolving $n/2(2p-1) + 1/(2p-1) = n$. This gives a value of $p = 3/4 + 1/2n \approx 3/4$, for large n.

The other stopping condition also applies when there is a majority which means that we would expect to never remove a ball from the bucket and simply traverse the list until the number remaining in the list becomes smaller than the number in the bucket. The expected size of the bucket is $n(2p-1)$ after the first phase and the expected length of the list is $2n(1-p)$. This means that we would expect there to be $n(3-4p)$ more balls in the list than the bucket and hence we would expect to need $n(3-4p)/2$ comparisons in the second phase.

This condition can only be triggered during the second phase which only happens when the first optimisation does not trigger. Since the first optimisation applies whenever $p \gtrapprox 0.75$ the third condition will only apply when p is between 0.5 and 0.75. Combining the analysis of the original algorithm with these two optimisations gives the following expected number of comparisons:

$$E[T] = \begin{cases} n-1+\dfrac{n}{2n(0.5-p)+2} & \text{if } p \leq 0.5 \\[2ex] n-1+\dfrac{n(3-4p)}{2} & \text{if } 0.5 < p \lessapprox 0.75 \\[2ex] \dfrac{n}{2(2p-1)} + \dfrac{1}{2p-1} & \text{otherwise} \end{cases} \tag{3}$$

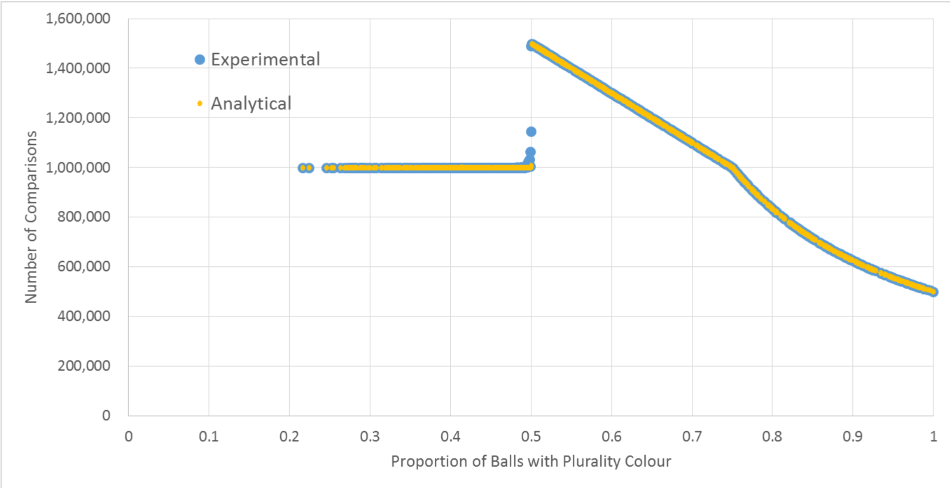

Figure 2 The Fischer-Salzberg algorithm has three distinct behaviours corresponding to its three stopping conditions which are shown in equation (3).

3.4 Experimental Validation

Fig. 2 shows the behaviour of the Fischer-Salzberg algorithm and validates our analysis. Three regions are evident in the experimental results corresponding to the three stopping conditions we have identified. The average discrepancy between our analysis and the observed results was 0.1% with a standard deviation of 1.1%.

4 Matula's Tournament Algorithm

4.1 Brief Description of the Algorithm

The Tournament algorithm, first proposed by Matula, is also based on the pairing off of balls but it stores the results of those comparisons for future use. The algorithm maintains a set of lists where each list has an associated weight ($w = 0, 1, 2, 3 \dots$) such that every item in the list is a shorthand for 2^w balls of the same colour. There is also a list of tuples of the form $\{v', v'', j\}$ where v' and v'' are balls of different colours both of which represent a list of 2^j balls of that colour.

Initially all balls are placed in the list with weight 0. Then pairs of balls from the same list are compared and either merged or discarded. If they match then they are merged and placed as a single ball in the next list. If they do not match then they are formed into a tuple and added to the list of discarded tuples. The process continues until no list contains more than one ball. At this point the candidate ball is the one in the most heavily weighted list which contains a ball (if none of the lists contain balls then there is no majority). A verification phase is needed in which the lists are traversed in reverse order and balls (really representing collections of balls) are compared to the candidate.

The pseudocode for the algorithm is in Appendix C.

4.2 Analysis of the Algorithm

The first phase of the Tournament algorithm must always run until no list contains more than one ball. The second phase stops when the counter reaches $n/2 + 1$.

The analysis of the Tournament algorithm is given in terms of recurrence relations with no closed form. Let $q_1, q_2 \dots q_k$ be the proportion of balls of colours $1, 2 \dots k$. Since balls are drawn at random we would expect to select two balls of the same colour, x, with probability approximately equal to q_x^2. In the first list (L_0) we have n balls and $n/2$ comparisons so that we would expect the number of balls in the second list, L_1, to be:

$$L_1 = n/2 \sum_{x=1}^{k} q_x^2$$

The proportion of balls with a given colour changes from list to list because colours which appear frequently in one list are more likely to get through to the next list whereas if a colour appears rarely then it is unlikely that two balls with that colour will have been paired in the previous list. Let $q_{x,i}$ be the proportion of balls of colour x in list i. We form the following recurrence relation for the proportion of a given colour in a given list:

$$q_{x,i+1} = q_{x,i}^2 / \sum_{x=1}^{k} q_{x,i}^2 \tag{4}$$

The number of balls expected in a given list, L_i, is also given by a recurrence relation:

$$L_{i+1} = L_i/2 \sum_{x=1}^{k} q_{x,i}^2 \tag{5}$$

These relations have no closed form but they can be applied systematically to derive the expected number of comparisons since for each list L_i we must perform $L_i/2$ comparisons. As for the second, verification, phase the expected number of comparisons depends on whether there is a majority or not. If there is no majority then we need to consider all the discarded tuples. The number of discarded tuples in the list can be derived from the size of each list:

$$D_i = \frac{L_{i-1} - 2L_i}{2}$$

For each tuple with a weight i there is a p_i chance that the tuple contains a ball of the candidate colour and a $(1 - p_i)$ chance that it does not because we assume that the candidate will be of the plurality colour. A tuple containing a matching ball will have an equal chance of it being the first ball in the tuple (requiring one comparison) or the second ball (requiring two comparisons). Other tuples always need two comparisons leading to the expected number of comparisons per tuple being $1.5p_i + 2(1 - p_i)$. If we examined all tuples during the verification phase then the expected number of comparisons during the second phase would be:

$$E[C_v] = \sum_i D_i \left(1.5p_i + 2(1 - p_i)\right)$$

When there is a majority we would expect the candidate counter to reach pn if we considered all the balls. By stopping when the counter reaches $n/2 + 1$ we would expect to save $pn - (n/2) - 1$ balls from being counted. We can give an upper-bound on the number of comparisons if we assume that all of the skipped balls are stored in the list of discarded tuples in tuples with weights of 0. These tuples all contain the candidate ball and so the expected number of comparisons per tuple is 1.5 leading to a total saving of $1.5(pn - (n/2) - 1)$ comparisons.

4.3 Optimisations and their Analsysis

As with the MJRTY algorithm, adding a counter of remaining balls to the verification phase allows the algorithm to end early if the candidate counter plus the number of remaining balls is less than $n/2 + 1$.

This improvement only applies when there is no majority. Assuming that the plurality colour ends up as the candidate, then after counting T balls we would expect the counter to be at Tp with $n - T$ balls remaining. Summing those values and setting to $n/2 + 1$ gives the expected number of balls that must be seen before stopping and therefore the number of saved balls which is $n - (n-2)/(2-2p)$. Here again we assume that all of these are in discarded tuples with weight 0 and so the number of tuples that do not need to be compared is half the number of balls saved. The expected number of comparisons for each of those saved tuples is $1.5p + 2(1-p)$. The overall expected number of comparisons in the second phase is therefore:

$$E[C_v] = \begin{cases} \sum_{i=0}^{log_2 n} D_i(1.5p_i + 2(1-p_i)) - 1/2[n - (n-2)/(2-2p)][1.5p + 2(1-p)] \\ \text{ if } p \leq 0.5 \\ \\ \sum_{i=0}^{log_2 n} D_i(1.5p_i + 2(1-p_i)) - 1.5(pn - (n/2) - 1) \\ \text{otherwise} \end{cases} \tag{6}$$

4.4 Experimental Validation

Fig. 3 shows three different regions for the Tournament algorithm and validates our analysis. Experimental results validate our analysis as the average discrepancy is 1.9% with a standard deviation of 1.9%. Note that again we have three regions, this is because when there is a majority the two phases involve a trade-off. The higher the value of p the fewer comparisons are needed in the second phase but the more are used in the first phase. After $p \gtrapprox 0.75$ there are virtually no comparisons in the second phase but the number from the first phase continues to increase leading to an overall increase in comparisons.

5 Conclusion

The majority problem is an interesting problem that is a special case of the heavy hitters problem. In this paper we have analysed the expected number of comparisons required by the three deterministic algorithms that appear in the literature, assuming an arbitrary number of colours. A direct comparison of the performance of the algorithms is show in Fig. 4.

It is interesting to note that the most-widely discussed of the algorithms (Boyer and Moore's MJRTY) algorithm requires the largest number of comparisons in the average case as well as the worst-case. In contrast, the Tournament algorithm requires the fewest comparisons for a large range of distributions. On the other hand, the MJRTY algorithm has very low overhead whereas the Tournament algorithm has significant overhead in terms of adding balls to lists.

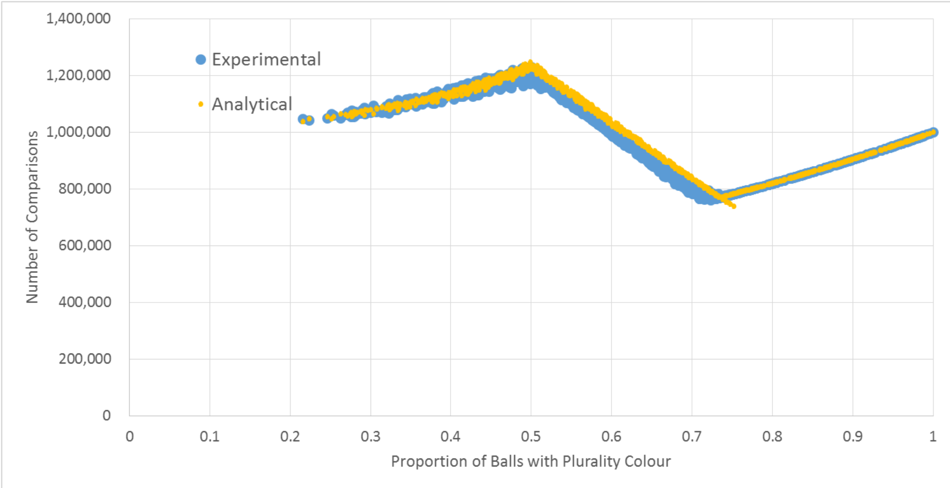

Figure 3 The Tournament algorithm has three different behaviours corresponding to the analysis in equation (6) with the third region ($p \gtrapprox 0.75$) being when there are virtually no comparisons in the second phase but an increasing number in the first.

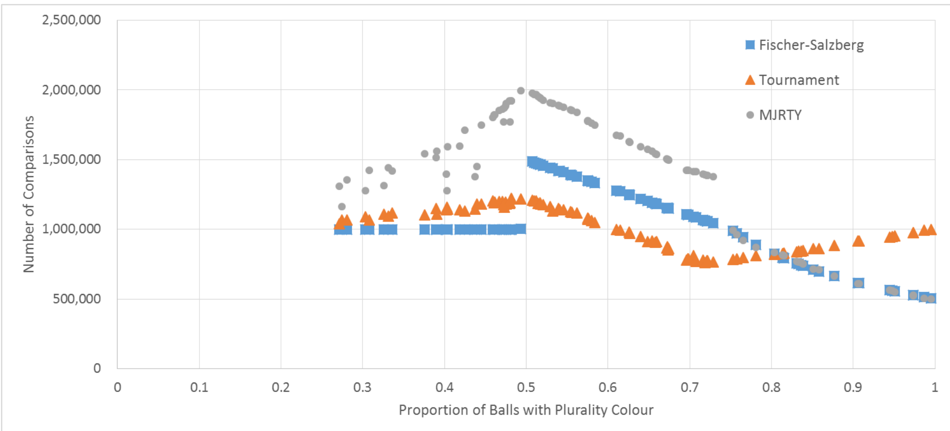

Figure 4 A comparison of the three algorithms shows that the much-discussed MJRTY algorithm typically requires the most comparisons.

Nevertheless, our results suggest that a fuller examination of the Tournament algorithm is warranted. There have been proposals for a parallel version of the MJRTY algorithm [8, 5, 6]. The Tournament algorithm, however, seems better suited for a parallel implementation and it would be interesting to see whether the overheads of parallelism result in the Tournament algorithm being more efficient than MJRTY.

The MJRTY algorithm has also been generalised for the heavy hitters problem, becoming the famous Misra-Gries algorithm [10]. Given that the Tournament algorithm requires fewer comparisons, it would be interesting to discover whether it could be successfully adapted for the heavy hitters problem as well.

Acknowledgement The author would like to thank Dr D.E. Rydeheard for his suggestions.

References

1 Laurent Alonso and Edward M Reingold. Analysis of boyer and moore's mjrty algorithm. *Information Processing Letters*, 113(13):495–497, 2013.

2 Laurent Alonso, Edward M Reingold, and René Schott. Determining the majority. *Information Processing Letters*, 47(5):253–255, 1993.

3 Laurent Alonso, Edward M Reingold, and René Schott. The average-case complexity of determining the majority. *SIAM Journal on Computing*, 26(1):1–14, 1997.

4 Robert S Boyer and J Strother Moore. *MJRTY—a fast majority vote algorithm*. Springer, 1991.

5 Massimo Cafaro and Piergiulio Tempesta. Finding the majority element in parallel. *In Submission*, 2010.

6 Massimo Cafaro and Piergiulio Tempesta. Finding frequent items in parallel. *Concurrency and Computation: Practice and Experience*, 23(15):1774–1788, 2011.

7 Michael J Fischer and Steven L Salzberg. Finding a majority among n votes. Technical Report 4, DTIC Document, 1982.

8 Chin-Laung Lei and Horng-Twu Liaw. Efficient parallel algorithms for finding the majority element. *J. Inf. Sci. Eng.*, 9(2):319–334, 1993.

9 DW Matula. An optimal algorithm for the majority problem. *Manuscript, Southern Methodist University, Texas*, 1990.

10 Jayadev Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

11 Michael E Saks and Michael Werman. On computing majority by comparisons. *Combinatorica*, 11(4):383–387, 1991.

12 Gábor Wiener. Search for a majority element. *Journal of statistical planning and inference*, 100(2):313–318, 2002.

A Pseudocode for MJRTY Algorithm

Algorithm 1 Pseudocode for the MJRTY Algorithm

```
c ← 0
for i ← 1 to n do
    if c = 0 then
        j ← i
        c ← 1
    else if x_i = x_j then
        c ← c + 1
    else
        c ← c − 1
    end if
end for
if c = 0 then
    No majority
else
    c ← 0
    for i ← 1 to n do
        if x_i = x_j then
            c ← c + 1
            if c > n/2 then
                Majority is of colour x_j
            end if
        end if
    end for
    No majority
end if
```

B Pseudocode for the Fischer-Salzberg Algorithm

Algorithm 2 Pseudocode for the Fischer-Salzberg Algorithm

```
l = 0
for i ← 1 to n do
    if x_i = list[l] then
        bucket.append(x_i)
    else
        list.append(x_i)
        l++
        if bucket.empty() then
            list.append(bucket.pop())
            l++
        end if
    end if
end for
C =list[l]
while !list.empty() do
    if list.pop()=C then
        list.pop()
    else
        if bucket.empty() then
            No majority
        else
            bucket.pop()
        end if
    end if
end while
Majority is C
```

C Pseudocode for the Tournament Algorithm

Algorithm 3 Pseudocode for the Tournament Algorithm

```
i ← 0
while ! list[i].empty() do
    j ← 0
    while j < list[i].length() && list[i].length() > 1 do
        if list[i][j] == list[i][j + 1] then
            list[i + 1].append(list[i][j])
        else
            tuples.add(list[i][j], list[i][j + 1], i)
        end if
        j ← j + 2
    end while
    i + +
end while
while i > 0 do
    if list[i].length() == 1 then
        if Candidate == NULL then
            Candidate = list[i][0]
            c = 2^i
        else
            if list[i][0] == Candidate then
                c ← c + 2^i
            end if
        end if
    end if
    i − −
end while
for all tuples do
    if tuple[0] == Candidate OR tuple[1] == Candidate then
        c ← c + 2^tuple[2]
    end if
end for
if c > n/2 then
    Candidate is majority
else
    No majority
end if
```